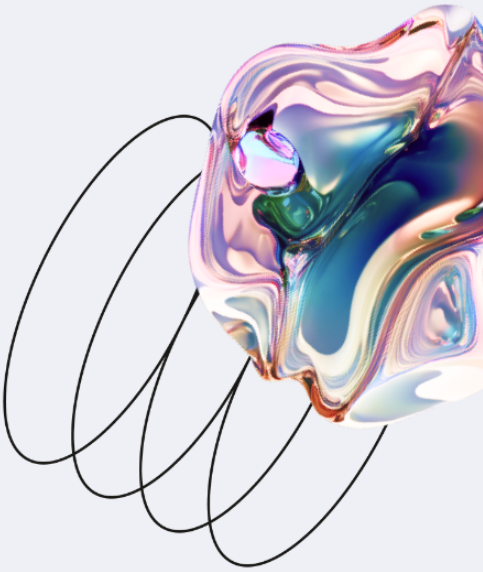
GeekBrains



Продвинутая работа с функциями и классами. Обработка ошибок

Продвинутый JavaScript

Оглавление

Введение

Ранее вы изучили базовое использования функций и классов. На предыдущем уроке мы рассмотрели итераторы, коллекции и модули. Сегодня мы будем обобщать полученные ранее знания и расширим их. Рассмотрим глобальный объект, кратко поговорим об объекте функции и расширим знания о стрелочных функциях.

Далее расширим знания о наследовании классов, вызове методов и о проверке класса.

В заключении урока нас ждёт глава о способах обработки ошибок в JavaScript. Мы изучим обработку ошибок со всеми возможностями, предоставляемыми самыми современными стандартами языка.

Глобальный объект. Объект функции

Глобальный объект

Глобальный объект — хранит функции и переменные, которые могут быть доступны в любой части программы.

По умолчанию — это встроенные объекты. Например, в браузере это объект window, а в Node.js — это объект global. Обратиться к объекту window можно как напрямую, так и используя его свойства self и frames:

```
1 console.log(window);
2 console.log(window.window);
3 console.log(self);
4 console.log(frames);
5 console.log(globalThis);
6 console.log(globalThis === window); // true
```

В последних стандартах введён новый глобальный объект globalThis, призванный стандартизировать обращение к глобальным объектам. В браузере он вернёт объект window.

В примере мы видим, что все обращения нам возвращают один объект. Более того, они строго равны между собой. Единственное отличие в том, что в Web Worker будет работать только с window.self.

> 💡 Web Worker — это механизм, позволяющий выполнять фоновые задачи отдельно от основного скрипта. Это полезная функция, но используется нечасто в узкоспециализированных скриптах, где требуются фоновые вычисления. В рамках нашего курса мы не будем рассматривать его подробнее.

Устаревший оператор объявления переменной var создавал переменные в глобальном объекте. Функции, объявленные с ключевым словом function, также создаются с привязкой к window.

```
var glob = 5;
function increment(val){
  return val+1;
}

console.log(window.glob); //5
console.log(window.increment); // ƒ increment(val){
//                                  return val+1;
//                                }
```

В связи с этим мы можем использовать поля и методы глобального объекта как переменные и функции, соответственно, не вызывая сам объект (мы и так всё время работаем в его контексте). Например, известные нам по предыдущим урокам функции alert(), confirm() и prompt() являются методами объекта window:

```
console.log(alert() === window.alert()); // true

alert('Можно так');
window.alert('А можно и так');
```

Объекты, объявленные в подключаемых модулях, не являются частью глобального объекта.

Переменные, созданные с помощью операторов let и const, а также функции, объявленные с помощью выражений (в том числе, анонимные), не являются методами глобального объекта:

```
const local = 5;
let localFunc = () => 'localFunc';

console.log(local); // 5
console.log(localFunc()); // localFunc

console.log(window.local); // undefined
console.log(window.localFunc); // undefined
```

При желании можно записать данные в локальный объект напрямую:

```
window.local = 5;

console.log(window.local); // 5
```

Но, как мы знаем, лучше не хранить в глобальной области какие-либо данные, так как потом заметно сложнее отследить какие-либо ошибки в расчётах. Иногда крайне полезно хранить глобальный объект для использования в приложении, а в некоторых фреймворках, например, в React, без него не обойтись. Для этого используют специальные правила, управляющие хранением данных, которые называются управлением состоянием приложения (State Management).

> 💡 Состоянием (State) приложения называются все данные приложения на этот момент. Данные эти могут храниться как в глобальной области, так и в одном из хранилищ.

О хранилищах мы поговорим на следующем уроке. Об управлении состоянием можно узнать в курсах фреймворков.

Отмечу, что глобальный объект хранит также все встроенные классы в JavaScript. На предыдущем уроке мы использовали класс Array и Symbol, их описание хранится именно в window:

```
1 console.log(window.Symbol.for); // ƒ for() { [native code] }
2 console.log(window.Symbol.iterator); // Symbol(Symbol.iterator)
3 console.log(window.Array.from); // ƒ from() { [native code] }
```

Объект функции

> 💡 К типу Object относятся не только функции, но и другие непримитивные объекты языка. Например, массивы, классы, промисы, события и остальные.

Все используемые в JavaScript элементы, имеющие значение, имеют свой базовый тип. Всего в JavaScript восемь типов: семь примитивов и Object. Значение какого типа имеют функции? Конечно же, Object.

Функциям присущи операции, которые можно проводить с объектами: передавать по ссылке, добавлять свойства, методы и тому подобное.

У функций есть два специальных свойства: name, с помощью которого мы можем получить имя функции и length, позволяющее получить количество аргументов, описанных при объявлении без учёта остаточных аргументов. Кроме того, мы можем определить собственные свойства.

Свойство name

Свойство name можно использовать для идентификации функции в средствах отладки или в сообщениях об ошибках. Оно доступно только для чтения и не может быть изменено оператором присваивания:

```
1 function someFunction() {}
2
3 someFunction.name = 'otherFunction';
4 console.log(someFunction.name); // someFunction
```

Если переменной присвоить анонимную функцию, имя этой переменной используется в качестве свойства name.

```
const someFunction = function someFunctionName() {};
someFunction.name; // "someFunctionName"
```

Выражения анонимных функций, созданные с использованием ключевого слова functionили стрелочных функций, будут иметь «»(пустую строку) в качестве имени.

```
(function () {}).name; // ""
(() => {}).name; // ""
```

Свойство length

Свойство length указывает, сколько аргументов ожидает функция, т. е. количество формальных параметров. Это число не включает остаточные параметры и включает только параметры перед первым параметром со значением по умолчанию. Напротив, arguments.length является локальным для функции и показывает количество аргументов, фактически переданных функции.

> 💡 Массив arguments — это локальный массив, доступный во всех функциях, кроме стрелочных. В нём записан каждый аргумент, с которым была вызвана функция, начиная с 0.

```
function func1(a, b, c) {
  console.log(arguments.length); // 3
}

func1(1, 2, 3);
```

Особенности стрелочных функций

Многие рассматривают стрелочные функции как синтаксический сахар для написания анонимных функций-выражений. В большинстве случаев это так, но есть три особенности стрелочных функций, что делает их особенными.

- Стрелочные функции не имеют своего this
- В стрелочных функциях отсутствует массив arguments
- Для однострочных стрелочных функций без выражений в фигурных скобках автоматически подставляется return для выражения

Рассмотрим пример:

```
1 const sum = (a, b) ⇒ a + b;
2
3 console.log(sum.length);
4
5 sum(1, 2);
```

Здесь мы записали простую стрелочную функцию и её вызов. Выражение после стрелки у нас автоматически стало возвращаемым. Если бы записали это выражение в фигурных скобках, для получения значения пришлось бы перед выражением добавить return.

Из предыдущей главы мы знаем, что массив arguments нам недоступен. Зато мы для получения количества аргументов можем использовать свойство length, оно отлично сработает.

Стрелочные функции не имеют своего контекста выполнения и при использовании this внутри стрелочной функции будет возвращаться undefined. При попытке привязки контекста стрелочной функции через метод bind(this) будет браться контекст выше, что непременно приведёт к ошибкам. Стрелочные функции также не могут быть вызваны как конструкторы с оператором new.

Защищённые и приватные свойства класса

Теперь поговорим о расширенных возможностях классов, которые появились в новых стандартах ECMAScript.

Защищённые свойства

Рассмотрим пример создания автомобиля:

```
class AutoMobile {
    _horsePowers = 0;
    set horsePowers(value) {
      if (value < 0) throw new Error("Отрицательное количество сил");
      this._horsePowers= value;
    }
    get horsePowers() {
      return this._horsePowers;
    }
    constructor(power) {
      this._horsePowers = power;
    }
}

// создаём новую машину
let auto = new AutoMobile(100);

// устанавливаем количество сил
auto.horsePowers = -10; // Error: Отрицательное количество сил
```

Здесь мы создаём новый автомобиль и задаём свойственное ему количество лошадиных сил. Для этого ему задаём сеттер и геттер. Свойство _horsePowers написали со знаком нижнего подчёркивания для обозначения, что свойство защищённое. Это не требование языка, а, так называемое, джентльменское соглашение между программистами, чтобы отличать такие свойства.

В сеттере при вводе отрицательного значения генерируется ошибка. Обработку ошибок обсудим чуть позже в этом уроке.

При установке положительного значения через обращение к свойству установится это значение и ошибок не будет.

Уберём сеттер и посмотрим, что получиться при попытке изменить свойство с мощностью автомобиля:

```
class AutoMobile {
    _horsePowers = 0;
    /* set horsePowers(value) {
      if (value < 0) throw new Error("Отрицательное количество сил");
      this._horsePowers= value;
    }  */
    get horsePowers() {
      return this._horsePowers;
    }
    constructor(power) {
      this._horsePowers = power;
    }
}

// создаём новую машину
let auto = new AutoMobile(100);

console.log(auto.horsePowers); // 100
// устанавливаем количество сил
auto.horsePowers = 10; // Uncaught TypeError: Cannot set property horsePowers of
#<AutoMobile> which has only a getter
```

Приватные свойства

В стандарте ECMAScript 2022 были введены приватные свойства класса. Эти свойства начинаются со знака # и имеют защиту на уровне языка.

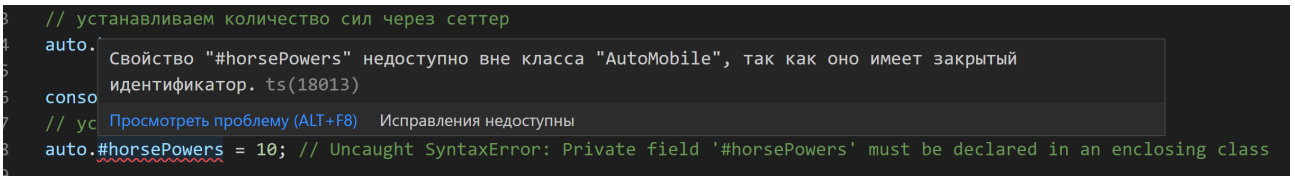
```
class AutoMobile {
    #horsePowers = 0;
    set horsePowers(value) {
      if (value < 0) throw new Error("Отрицательное количество сил");
      this.#horsePowers= value;
    }
    get horsePowers() {
      return this.#horsePowers;
    }
    constructor(power) {
      this.#horsePowers = power;
    }
}

// создаём новую машину
let auto = new AutoMobile(100);

// устанавливаем количество сил через сеттер
auto.horsePowers = 50;

console.log(auto.horsePowers); // 50
// устанавливаем количество сил напрямую
auto.#horsePowers = 10; // Uncaught SyntaxError: Private field '#horsePowers'
  must be declared in an enclosing class
```

Устанавливать и читать приватные свойства получится только через сеттер и геттер. При обращении напрямую нам сгенерируется ошибка. Более того, продвинутые IDE, как, в моём случае, VS Code, нам сообщит об ошибке сразу:



Проверка класса оператором instanceof

Ранее мы проверяли тип переменной оператором typeof. Он отлично работает с определением типа примитива. Но что нам делать, если нужно проверить, к какому классу относится объект?

```
let arr = new Array(1,2);

console.log(arr); // (2) [1, 2]
console.log(typeof arr); // object
```

Тут мы создали массив из двух элементов и пытаемся определить его тип с помощью typeof. Он нам выдал значение object, и это правильный результат. Ранее мы обсуждали, что массивы относятся к объектам. Как нам понять, что в итоге перед нами массив, а не просто объект. Для этого нам поможет оператор instanceof:

```
let arr = new Array(1,2);

console.log(arr); // (2) [1, 2]
console.log(arr instanceof Array); // true
```

Он сравнивает тип переменной-объекта с предложенным и выдаёт булево значение, которое отвечает, совпадают ли их типы.

Оператор instanceof нам поможет для определения

- Принадлежит ли объект к предлагаемому классу
- Либо к одному из встроенных классов, как в примере с Array
- Или для функций — конструкторов:

```
function Rabbit() {}

console.log( new Rabbit() instanceof Rabbit ); // true
```

Опциональная цепочка вызовов методов и оператор нулевого слияния

Это две новые возможности, которые нам добавились в стандарте ECMAScript 2020. Они позволяют нам избегать дополнительных проверок на null и undefined, что делает наш код понятнее и чище.

Опциональная цепочка применяется для методов объектов, которые в определённый момент времени могут быть undefined или null. Например, есть у нас асинхронная операция, которая нам выдаст результатом объект типа:

```
const obj = {
    key: 'value'
}
```

Пока ещё не пришли данные, вызов объекта obj нам выдаст значение undefined. При вызове свойства key нам прилетит ошибка:

```
let obj = {
    key: 'value'
}

obj = undefined;

console.log(obj.key); // Uncaught TypeError: Cannot read properties of undefined
 (reading 'key')
```

Чтобы избежать этой ошибки, требовалось сделать проверки на undefined вышестоящих методов объектов и сам объект. Эти проверки нам может заменить оператор опциональной цепочки:

```
let obj = {
    key: 'value'
}
obj = undefined;
console.log(obj?.key); // undefined
```

В итоге, никакой ошибки выдано не будет, нам просто вернётся значение undefined без лишних нагромождений условных операторов.

Для тех же целей введён в стандарт оператор нулевого слияния. Допустим, есть у нас две переменных a и b. Если значение в переменной a равно null или undefined, мы хотим использовать значение переменной b. Мы можем назвать переменную b значением по умолчанию, если в переменной a нет значения.

```
let a = null;
let b = 'значение по умолчанию';

console.log(a ?? b); // значение по умолчанию

a = 1;
console.log(a ?? b); // 1
```

Мы можем использовать цепочки из операторов нулевого слияния для проверки нескольких переменных и установленного значения по умолчанию:

```
let a = null;
let b = undefined;

console.log(a ?? b ?? 'значение по умолчанию'); // значение по умолчанию

```

```
6 b = 1;
7 console.log(a ?? b ?? 'значение по умолчанию'); // 1
```

Этот оператор является синтаксическим сахаром для следующего выражения:

```
1 let result = (a === null || a === undefined) ? b : a;
```

Где в переменную result нам записывается то, что до этого выводилось в console.log

🔥 Ранее часто пользовались логическим оператором ||, но он мог отнести к отсутствию данных значения, которые при неявном преобразовании типов приводились к булеву значению false: пустую строку, 0, NaN и, собственно, false.

Обработка ошибок в JavaScript

Цепочка try, catch, finally

Обработка ошибок в JavaScript по синтаксису похожа на ту же операцию в других языках программирования — в PHP, C и других.

Какой-то блок кода оборачивается в try{}, где мы ловим ошибки. Отмечу, что синтаксические ошибки не будут найдены обработчиком ошибок, так как скрипт просто не выполнится и до завершения работы обработчика дело не дойдёт. Обработчик также не распознаёт асинхронные ошибки, так как работает синхронно и завершит свою работу прежде, чем придёт асинхронная ошибка.

Синтаксис цепочки такой: после просматриваемого блока, обёрнутого в try, идёт опциональный блок catch, в параметре которого идёт объект ошибки. В стандарте ECMAScript 2019 объект ошибки можно не указывать, если он не требуется. Далее идёт опциональный блок finally, код в котором выполнится после обработки кода с блоком try и обработкой ошибок, если они были. Один из блоков catch или finally должен обязательно присутствовать:

```
1 try{
2     undefined = 1;
```

```
3 }
4 catch(err){
5     console.log(err);// TypeError: Cannot assign to read only property
  'undefined' of object '#<Window>'
6 }
```

При этом скрипт продолжит выполняться и не упадёт в ошибку. Пример с пустым catch и с блоком finally:

```
1 try{
2     undefined = 1;
3 }
4 catch{
5     console.log('Что-то произошло');
6 }
7 finally{
8     console.log('Но жизнь продолжается');
9 }
```

Есть пара нюансов.

- Переменные, объявленные в блоке try, не будут видны в catch и finally. Поэтому, если нужен обмен данными между ними, лучше объявить их на уровень выше.
- Код в блоке finally выполнится в любом случае, даже если в блоке try был оператор return.

Пользовательские ошибки

Для отладки кода мы можем генерировать свои ошибки с помощью оператора throw.

```
1 throw 'error'; // Uncaught error
```

Естественно, лучше это делать внутри блока try, чтобы у нас не остановилось выполнение скрипта. В качестве параметра оператор throw может принимать любые значения, но чаще всего отправляют встроенный класс Error глобального объекта:

```
try{
    throw new Error('it`s error!');
}
catch(err){
    console.log(err);
    console.log(err.name);
}
```

Если мы не знаем, как обработать ошибку в текущем контексте, мы можем пробросить ошибку на уровень выше, поместив throw внутрь блока catch:

```
try{
    throw new Error('it`s error!');
}
catch(err){
    if(!(err instanceof Error)) throw new Error('Непредвиденная ошибка');
}
```

Мы также можем расширить класс Error с помощью наследования для придания ему каких-то своих необходимых свойств.

В стандарте ECMAScript 2022 появилась возможность указывать причину возникновения ошибки, которой мы тоже при желании можем воспользоваться при расширении класса Error.

Вообще, необходимость самостоятельно расширять класс Error появляется нечасто. Во многих библиотеках имеется своё расширение класса Error для присутствующих в ней объектов, функциональности которых для повседневной работы достаточно.

Подведём итоги

В этом уроке мы разобрали продвинутую работу с функциями и классами, которая соответствует самым современным стандартам языка. Стандарты постоянно совершенствуются, это меняет подходы к написанию многих привычных нам вещей. Стрелочные функции очень широко используются в колбэках, которые, в свою очередь, широко применяются во встроенных методах и в промисах. О промисах мы поговорим на следующем уроке.

Кроме этого, разобрали работу с ошибками, их обработку и отладку кода, когда ошибки возникают. Это нам поможет повысить доработку кода и качество разрабатываемого продукта.